Ariane + NVDLA: Seamless Third-Party IP Integration with ESP

Davide Giri, Kuan-Lin Chiu, Guy Eichler, Paolo Mantovani, Nandhini Chandramoorthy[‡]
and Luca P. Carloni
{davide_giri,chiu,guyeichler,paolo,luca}@cs.columbia.edu,nandhini.chandramoorthy@ibm.com
Department of Computer Science, Columbia University in the City of New York, New York, NY 10027
[‡]IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598

ABSTRACT

The growth of the RISC-V movement and the demand for specialized hardware have fueled a proliferation of open-source processors and accelerators. Leveraging the large amount of available IP components requires the capabilities for integrating them effectively in a system-on-chip. We address this goal by augmenting ESP, an open-source platform for agile design of heterogeneous SoC architectures. Since its release, ESP has provided a set of different flows to design new accelerators with different specification languages and automate their SoC integration. We add support for the seamless integration of third-party accelerators by developing a new type of interface that retains the benefits of the ESP platform services. We demonstrate these capabilities by showing how to integrate the Ariane RISC-V core with multiple instances of the NVIDIA Deep Learning Accelerator into an SoC architecture that leverages a scalable memory hierarchy and network-on-chip. The new contributions, which are already available in the ESP release, allow designers to rapidly prototype complex SoC architectures on FPGAs with a push-button design flow.

1 INTRODUCTION

Hardware accelerators are pervasive across a multitude of SoC architectures that integrate many heterogeneous components [25]. As the complexity of the design effort keeps growing with each SoC generation, the addition of new capabilities is increasingly limited by the engineering effort and team sizes [30].

Thanks to the *open-source hardware (OSH)* movement, which largely stemmed from the RISC-V project [4, 23], a growing community of researchers and engineers is contributing to the proliferation of OSH processor cores and accelerators [24]. Made available in the public realm, these intellectual property (IP) blocks allow designers to exploit the aggregate expertise of the entire community when realizing a new SoC. However, leveraging a large amount of IP components and combining them into a solution for a target domain remains a challenging task that requires the capabilities for integrating them effectively into an SoC architecture.

To tackle this challenge we augment ESP, an open-source platform for the agile design of heterogeneous SoCs [14]. ESP combines an architecture and a methodology. The scalable tile-based architecture simplifies the integration of heterogeneous components. The flexible methodology embraces the use of a variety of design flows for component development. In particular, these flows simplify the design of new loosely coupled accelerators [15] and their integration into the architecture. Users can choose to specify a new accelerator at different abstraction levels, including cycle-accurate RTL descriptions with SystemVerilog or Chisel, loosely-timed or untimed behavioral descriptions with SystemC or C/C++, and domain-specific languages such as Keras TensorFlow, PyTorch and ONNX for the application domain of embedded machine learning [19].

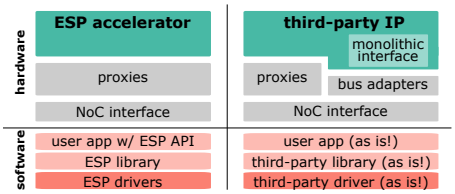

Fig. 1: Native and third-party ESP accelerator integration.

Each of these specifications can be synthesized into an implementation and evaluated in an FPGA-based prototype of the SoC.

With this work, we add support for the integration of third-party accelerators into ESP by developing a new type of *socket interface* that retains the benefits of the ESP *platform services* [11]. The new *third-party flow (TPF)* allows any user of ESP to integrate an existing accelerator by simply choosing among a set of bus-standard adapters that interface it with the NoC connecting the tiles in the architecture. The new TPF socket enables the seamless integration of a loosely coupled accelerator such that designers can execute the original software application *as is*. The only requirement is that the application and the device drivers can be compiled for RISC-V.

Fig. 1 captures the differences between integrating an ESP accelerator and a third-party IP with the new TPF. The two top quadrants show the main features of the two accelerator sockets. Notably, the network interface is the same, granting both accelerators access to the ESP platform services. However, some of the building blocks of the ESP socket, which are proxies for these services, are replaced by simpler adapters between the NoC and a standard bus interface. Generic accelerator datapaths, in fact, are not typically decoupled from their system interface, which is often part of a monolithic IP that behaves according to a bus protocol. The two bottom quadrants, instead, show the corresponding software stacks. Thanks to the new TPF, an ESP instance can drive a third-party accelerator by simply running its original software and device driver.

We demonstrate these new capabilities by integrating the Ariane 64-bit RISC-V core [1, 45] and multiple instances of the NVIDIA Deep Learning Accelerator (NVDLA) [37] into an SoC instance, which we implement on an FPGA board by means of the ESP push-button flow for rapid prototyping. This SoC instance can use up to four DDR channels to external memory, reconfigure at run-time the cache hierarchy, offload multiple concurrent workloads to the NVDLA instances, and monitor the effect on the NoC traffic and the memory hierarchy in real time.

The contributions described in this paper are already available as part of the ESP release [14]. Designers can use the new TPF to rapidly prototype RISC-V-based SoCs that combine a mix of their own accelerators with other OSH components in order to run more efficiently their software for the target application domain.

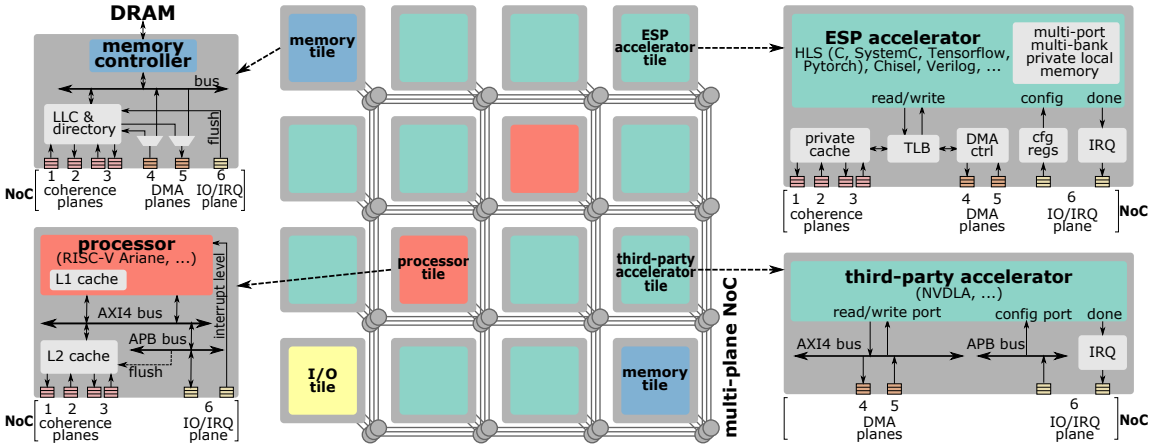

Fig. 2: Example of ESP architecture with 16 heterogeneous tiles interconnected by a multi-plane NoC. The NoC is the core of the ESP architecture, providing inter-tile communication and supporting the system-level platform services.

2 THE ESP ARCHITECTURE

The ESP architecture is structured as a tile grid [11]. For a given application domain, the architect decides the structure of the SoC by determining the number and mix of tiles with the help of the ESP graphical user interface. For example, Fig. 2 shows an SoC instance with 16 tiles organized in a 4×4 grid. There are four main types of tiles: processor tile, accelerator tile, memory tile for the communication with main memory, and auxiliary tile for peripherals (e.g. UART or Ethernet) or system utilities (e.g. the interrupt controller). In the first version of ESP, each processor tile contained a 32-bit LEON3 SPARC core [13] and each accelerator tile contained a loosely-coupled accelerator designed with the ESP methodology. Thanks to the contributions described in this paper, now ESP allows designers to choose between this LEON3 core and the 64-bit Ariane RISC-V core [45] for a processor tile as well as to instance a third-party IP block within any accelerator tile.

Each tile is encapsulated into a *modular socket* (aka *shell*) that interfaces it to a network-on-chip (NoC). In addition, the socket implements a set of *platform services* that provide pre-validated solutions for many important SoC operations, including: accelerators configuration, memory management, sharing of SoC resources, and dynamic voltage and frequency scaling (DVFS). For example, the coherence model of each accelerator can be reconfigured dynamically based on the particular workload [20]. Dynamic voltage and frequency scaling can be applied at the granularity of each tile [34]. This fine granularity applies also for such services as performance counters and operation monitors. The platform services is one of the keys to rapid prototyping of full SoCs. At design time, it is possible to choose the combination of services for each tile. At runtime, many of these services offer reconfigurability options.

The NoC interacts only with the socket to provide on-chip communication and support the platform services. In the current version of ESP, the NoC has a packet-switched 2D-mesh topology with multiple physical planes. These allow the NoC not only to prevent various types of protocol deadlock, but also to distribute messages to maximize the performance of processors and accelerators.

Processor Tile. Both the LEON3 and the Ariane RISC-V processor can run Linux and come with their private L1 caches. The instance of the modular socket for the processor tile augments them with a private unified L2 cache of configurable size. The processor integration into the distributed ESP system is transparent, i.e. no ESP-specific software patches are needed to boot Linux. A MESI directory-based protocol provides support for system-level coherence on top of three dedicated planes in the NoC [21]. Another plane supports I/O and interrupt-request channels that are used for various purposes, including accelerator management.

Memory Tile. Each memory tile contains a DDR channel to external DRAM. The number of memory tiles can be configured at design time and typically varies from one to four depending on the size and type of the given SoC. All necessary hardware logic to support the partitioning of the addressable memory space is automatically generated. This logic is also completely transparent to software. Hence, each memory tile contains a partition of configurable size of the last-level cache (LLC) and the corresponding directory. The directory implements a NoC-based MESI protocol extended with support for coherent-DMA transfers for the accelerators [21]. The MESI protocol messages are routed through the three coherence planes. The DMA packets are routed on two additional dedicated planes. The memory tile routes non-coherent DMA requests directly to main memory, bypassing the cache hierarchy, while the coherent DMA requests are sent to the directory.

Accelerator Tile. Each accelerator tile contains the specialized hardware for a loosely-coupled accelerator that executes a coarse-grained task [15]. Once an application running on a processor core has acquired and configured a given accelerator, its execution happens independently from the core while exchanging (typically very large) data sets with the memory hierarchy [33].

At design time, the modular socket decouples the design of the accelerator from the design of the NoC and the rest of the SoC, and provides pre-designed implementations for a set of accelerator-independent platform services. In particular, the services relieve the accelerator designer from the burden of "reinventing the wheel" with respect to implementing various mechanisms such as: accelerator configuration through memory-mapped registers, virtual memory and DMA services for data transfer with the accelerator's private local memory, and interrupt requests for interactions with

the processor cores. The socket of the accelerator tile supports also multiple cache-coherence models and it enables accelerator-to-accelerator communication, such that DMA transactions are either routed to memory (shared memory communication), or directly to other accelerators (accelerator-to-accelerator communication).

The concept of socket plays a key role in supporting the flexibility of the ESP methodology in the sense that it accommodates accelerators designed with many different design flows, as mentioned in the introduction. Furthermore, sockets have been instrumental to adding support for the integration of third-party accelerators like NVDLA. Fig. 2 contrasts the socket for a newly-designed accelerator (top right corner) with the one for third-party accelerators (bottom right corner). Section 3 discusses in detail the difference of integrating these two types of accelerators in ESP.

Accelerator Programming. ESP comes with a software stack and an application programming interface (API). The ESP accelerator API library simplifies the invocation of accelerators from a user application, by exposing only three functions to the programmer [19]. Underneath, the API invokes the accelerators through the Linux device drivers, which are automatically generated for ESP accelerators. The lightweight API can be easily targeted from existing applications or by a compiler.

Fig. 3 illustrates the ESP software stack through the (simplified) case of an application with five kernels, two executed in software and three implemented with an accelerator. For a given application, the software execution of a computationally intensive kernel can be replaced with a hardware accelerator by means of a single function call (`esp_run()`). Thanks to the `esp_alloc()` and `esp_cleanup()` functions, data can be shared between accelerators and processors in such a way that *no data copies are necessary*. Data are allocated in an efficient way to improve the accelerator's access to memory without compromising the software's performance [33].

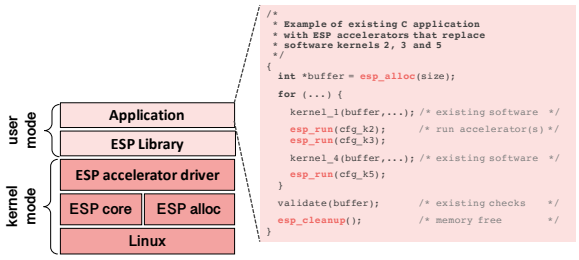

Fig. 3: ESP software stack for accelerator programming.

3 ACCELERATOR INTEGRATION

In order to explain how third-party accelerators can be integrated in ESP, we first describe the native ESP accelerator flow. Then, we show how the ESP accelerator socket can collapse into a third-party accelerator socket that offers access to the system's shared resources through standard bus interfaces.

3.1 ESP Accelerator Flow

The ESP accelerator flow consists of a sequence of automated steps. The process begins by launching an interactive script that asks the user to enter the following application-specific information required to generate a *skeleton* of the accelerator specification:

• **Accelerator name and ID:** a unique pair of name and device ID for the new accelerator.

• **Design flow:** the desired high-level synthesis (HLS) tool. The current choice is among Cadence Stratus HLS [41], Xilinx Vivado HLS [44] and the open-source project HLS4ML [17, 42].

• **Accelerator registers:** the list of user-defined configuration registers and value ranges. Each entry translates into the generation of a 32-bit configuration register in the accelerator socket; the register is exposed to software through the addition of a field into the *descriptor* data structure.

• **Data bit-width:** the default size of the data word for the accelerator. This can be one byte (8), half word (16), word (32), or double word (64). This information is used to generate examples of data transaction requests in the ESP accelerator skeleton.

• **Input data size:** the size of the input data set. This entry can be specified as function of the configuration registers.

• **Output data size:** the size of the output data set. This can be specified as a function of the configuration registers as well.

• **Chunk factor:** integer divisor of the input and output data sets. This parameter determines the size of the input and output data tokens, i.e. the portion of input data to be loaded before starting computation and the corresponding portion of output results stored back to memory by the accelerator. At each invocation, an ESP accelerator typically loads several input data tokens from DRAM, it processes them, and stores several output data tokens back into DRAM. The three phases of load, compute and store are pipelined, thanks to the accelerator's *private-local memory* (PLM) [40]. The capacity and the micro-architecture of the PLM depend on the values entered during this initial configuration phase.

• **Batch factor:** integer multiplier of the input and output data set. This parameter can be a function of the configuration registers and determines the number of complete input data sets that get processed in a single invocation of the accelerator. While the chunk factor sets the size of the PLM (in combination with input and output data size), the batch factor has no impact on local storage.

• **In place:** flag specifying whether the store phase should compute an offset for write transactions, or assume that output results can overwrite input data in main memory.

The ESP accelerator initialization script generates an HLS-ready skeleton of the accelerator specification together with the unit testbench, HLS and simulation scripts, bare-metal driver, Linux driver, and a sample testing application. When the user selects either the Stratus HLS or Vivado HLS flow, this skeleton is generated as HLS-synthesizable SystemC or C++ code, respectively. The designer must complete the specification by adding the accelerator-specific code for the computation phase to the skeleton and, possibly, by modifying the generated code to account for any irregular memory transactions. The resulting specification is the HLS-ready code, which is the entry point of the design automation process for both the Stratus HLS and Vivado HLS flows. For the HLS4ML flow, instead, the entry point is a trained machine-learning model created with Keras [29], ONNX [6] or Pytorch [38], while the HLS-ready code is automatically generated as a complete synthesizable C++ specification of the accelerator.

Besides the accelerator specification, ESP users are responsible for adding application-specific code to the generated unit testbench.

This code should read (or generate) input data and compute the corresponding golden output data for verification. The code added to the C++ or SystemC testbench must be copied into the bare metal and Linux applications as well. These applications replace the role of the testbench for system-level simulation and FPGA prototyping.

3.2 ESP SoC Flow

After completing the HLS-ready code, all types of ESP accelerator flow follow the same automated steps, regardless of the particular language chosen for the specification. The main steps are:

1. High-level synthesis. Run the chosen HLS tool to generate one or multiple RTL implementations of the accelerator. The generated Verilog code is added to the ESP library of IP blocks for integration. During this step, ESP users can perform a design-space exploration for the new accelerator by tuning the HLS directives and obtain several functionally-equivalent RTL implementations, each with a different cost vs. performance trade-off point [32, 39]. Thanks to the ESP socket, any of these RTL implementations can be integrated in the SoC, regardless of its throughput and latency [35]. This approach to digital design that combines latency-insensitive interfaces [10, 12] with HLS is raising interests in the silicon industry, where companies seek paths to more agile design flows for SoCs [30, 46].

2. Accelerator verification. The unit testbench stimulates the accelerator implementations and monitors their outputs. Since the ESP accelerator socket is modeled by the testbench, the completion of this step indicates that the integration of the accelerator is correct and the accelerator is ready to be deployed in the SoC.

3. Software build. For each accelerator, ESP automates building both bare-metal binary and Linux image for testing. The bare-metal driver consists of a loadable file for the external memory available on the target FPGA development board. In addition, a standard *srec* text file is dumped for the memory model used in simulation.

4. SoC configuration. A simple graphical user interface (GUI) helps designers create an ESP configuration file by selecting where the new accelerators should be located in the tiles grid, together with processor cores and memory tiles.

5. System-level simulation. For each target FPGA board, ESP provides a testbench to simulate the complete execution of a bare-metal program, including boot loader and interaction with peripherals. Users can specify the target program to be the bare-metal driver for the new accelerator, thereby testing how the accelerator behaves when driven by an ESP processor tile.

6. FPGA prototyping. When targeting one of the supported FPGA boards, ESP users can prototype their SoC without prior FPGA experience. The generation of the bitstream file, the programming script and the deployment of software are fully automated. The ESP instance is controlled through an Ethernet interface that allows quick loading of programs into main memory, updating the boot loader into an on-chip RAM and resetting the processors.

3.3 Third Party Socket

The ESP flows described above rely on a set of modular components integrated with latency-insensitive interfaces. These are cache controllers, adapters, arbiters and decoders that act as proxies for the shared system resources. Any accelerator, regardless of its particular microarchitecture, needs access to the same resources: a minimal list includes memory, configuration-status registers (CSR) exposed to software, and interrupt delivery. Without using an automated flow, like the one offered by ESP, designers are typically responsible for implementing the logic to interface with the SoC and access such resources. Even when leveraging corporate CAD tools that facilitate the integration of accelerators into their proprietary systems, this logic is generated as part of the new IP block. For instance, this applies to the well-known Xilinx Vivado tool that combines HLS with an IP integrator flow specific for ZYNQ SoCs [44].

In general, the obvious choice for designers, is to pick one or more standards to interface with a hypothetical SoC. Among the available options, the open standards *AXI*, *AHB* and *APB* from ARM are the most widely adopted [2]. Other popular alternatives include *TileLink* from UC Berkeley [9], *Avalon* from Intel [27], *CoreConnect* from IBM [26] and *Wishbone* from the OpenCores community [43].

For the new TPF, we implemented adapters for ARM-based IP blocks. Specifically, AXI, AXI-Lite and APB adapters are now part of the open-source release of ESP. We anticipate implementing adapters for other standards in the near future, thus expanding the set of third-party IP blocks that ESP can seamlessly integrate.

The idea behind the TPF for ESP is simple: we modify the ESP socket by relying on its modularity so that it "collapses" into a set of bus-standard interfaces between a generic accelerator and the ESP NoC, as illustrated by the block diagram of two accelerator tiles in Fig. 2. First, we replace the DMA engine with an AXI master port that converts AXI transactions into NoC packets. From the accelerator designer viewpoint, this new proxy takes the role of the AXI crossbar that is commonly used for integration on a traditional bus-based SoC. Since NoC packets can be routed to both non-coherent and coherent DMA planes, third-party accelerators can still benefit from run-time selection of the coherence model. We remove the optional private L2 cache and the translation-lookaside buffer. The former, when is needed, is usually part of the third-party IP, while the latter is bypassed by the AXI master interface, which can issue requests to physical memory based on the internal behavior of the accelerator.

Similarly, we replace the CSR logic with an existing APB proxy interface. While registers are part of the native ESP accelerator socket, CSRs of a third-party accelerator are integrated into the accelerator design. Our APB proxy is used in ESP to connect the slave port of memory-mapped registers and devices with the NoC. This component can be reused *as is* to integrate a third-party accelerator. For flexibility, we add an optional adapter to convert APB transactions to AXI-Lite or full AXI slave packets.

Finally, we modify the proxy responsible for interrupt delivery so that it accepts an interrupt clear (CLR) message from the interrupt controller. This change is necessary to integrate accelerators, like NVDLA, that implement two common behaviors: (*a*) level-sensitive interrupt requests (IRQ) with no return to zero required; (*b*) overlapped execution of one task and configuration of the next one. In this scenario, when software occasionally slows down, due to a variable workload or to the non-deterministic behavior of the operating system, distributing interrupts over the NoC can lead to a deadlock condition. Consider the following steps: (*1*) the accelerator asserts and holds an IRQ; (*2*) the interrupt controller receives the message

from the interrupt proxy and another proxy on this tile sends a message to the processor tile; (*3*) the processor acknowledges the IRQ, saves the state of the interrupt controller, masks further interrupts and enters the interrupt handler routine; (*4*) the interrupt handler issues an accelerator-specific clear to the accelerator; (*5a*) upon receiving the clear, the accelerator may be ready to raise a second IRQ and can do so without first bringing the interrupt level to zero; (*5b*) alternatively, even if the interrupt level returns to zero, the new IRQ may reach the interrupt controller when interrupts are still masked; (*6*) the processor restores the state of the interrupt controller as it exits the interrupt handler routine. If either step (*5a*) or (*5b*) occurs, then the second IRQ message gets lost and the accelerator gets stuck. We solve this problem in a general way that applies not only to NVDLA. We make the proxy on the interrupt controller tile send a clear message to the accelerator tile when the processor restores the interrupts state. In this way, while the interrupt handler sends an accelerator-specific clear message, the interrupt controller proxy sends a generic clear message, informing the accelerator tile that a new interrupt can be correctly received.

3.4 Third Party Accelerator Flow

After implementing the third-party socket, we augment the ESP infrastructure to implement a new flow that users can leverage to integrate their existing accelerator IP blocks in a few simple steps:

1. Accelerator Definition. Fill in a short XML file with some key information. This includes a unique accelerator name and ID, the name of reset and clock signals, an optional prefix for the AXI interface signals, and the user-defined width of AXI control signals.

2. Source RTL. Create a list file for each type of source RTL, including Verilog, SystemVerilog, VHDL and VHDL packages.

3. Make. Create a Makefile with all targets that apply among RTL generation (*vmod*), linux device driver (*kmd*), user-space application (*umd*), bare-metal driver (*bmd*).

4. Software Objects. Create a list file for driver modules, software executable, libraries, and any other binary required by the user application.

5. RTL Wrapper. Write a Verilog top-level wrapper to assign any non-standard input port of the third-party accelerator (e.g. disable testmode, if present) and expose the AXI and APB interfaces for the new ESP socket. This last step mainly consists in attaching wires without implementing any logic. ESP users may look at the NVDLA integration example to evaluate the simplicity of implementing the RTL wrapper for the TPF.

After applying these steps, users can configure an instance of ESP, run simulations, and prototype their system on FPGA by following the SoC flow described in Section 3.2, starting from Step 3 (software build). The only notable difference is that the third-party software, including device drivers and applications, is compiled by leveraging the user-specific Makefile targets described above. For each IP, a new folder is generated in the Linux file system image. This folder is available inside root home directory after booting the ESP instance.

4 PROCESSOR INTEGRATION

We developed ESP with a holistic system view, rather than focusing on processor cores like other open-source SoC generators. The native ESP accelerator flow and the new TPF for accelerators are, in fact, the main distinguishing factors that make ESP particularly suitable for heterogeneous SoC design. Nevertheless, since processors retain a fundamental role in controlling the execution of any SoC, we made sure to extend the flexibility of ESP also to the choice of different processor cores. To do so, we designed the processor tile by leveraging a combination of the same proxy components and adapters that we use for the accelerator sockets. The resulting structure is illustrated in Fig. 2. The 64-bits RISC-V Ariane processor core from ETH Zurich [1] is transparently integrated in ESP through AXI master ports. These are part of the AXI proxy used in the third-party accelerator socket. ESP offers an AHB adapter as well to interface with the 32-bits SPARC-V8 Leon3 processor core from Cobham Gaisler [13]. Similarly, non-cacheable read and write operations are forwarded to the NoC with the APB adapter that is used across all ESP tiles to expose memory-mapped registers and devices to software. The L2 cache is just an instance of the optional private cache in the ESP accelerator socket. The write-back L2 cache and the companion LLC and directory splits, located in each memory tile, implement an extended MESI protocol that supports run-time reconfiguration of the accelerator coherence models [22]. The interrupt-level proxy, represented in Fig. 2 as a simple queue, is the only implementation-specific component in the ESP processor tile. This proxy is one half of a network adapter that allows processor cores to interact with the platform interrupt controller and the system timer. Both are located in the ESP miscellaneous I/O tile, which hosts all peripherals shared in the system (except from memory), i.e.: the Ethernet NIC, UART, a digital video interface and a debug link to control ESP prototypes on FPGA.

The interrupt-level proxy delivers single-flit packets over the NoC. In the case of RISC-V, these packets can only originate from the interrupt controller or the timer in the I/O tile and terminate at one processor tile. Interrupt claim, acknowledge or clear occur via memory-mapped register accesses. Conversely, when generating an ESP instance with the Leon3 core, interrupt-level request and acknowledge use a custom protocol that requires the proxy to send single-flit packets from processor tiles to the I/O tile as well. The payload in these packets depends on the implementation-specific protocol.

Thanks to the modularity of the ESP tiles, our team integrated Ariane in the span of a few weeks, while keeping the option of Leon3 as an alternative core. This is possible because we rely on ESP standard bus adapters and proxies to decouple all platform services from the particular third-party IP block to be integrated. The sole exception is the simple interrupt-level proxy, which requires a different implementation for each processor.

The resulting system can execute any RISC-V program *as is*, with no ESP-specific patches. This includes Linux, the bootloader of Ariane and third-party device drivers, such as the NVDLA runtime.

Given the presence of implementation-dependent key components, such as the interrupt controller, integrating different third-party processor cores cannot be automated. Nevertheless, the integration of Ariane proves that ESP still greatly facilitates the task. We envision to support more processor cores in the near future, e.g. by extending the set of proxy and adapters to TileLink and Wishbone. To name a popular example, supporting TileLink would allow ESP to seamlessly integrate instances of the RISC-V processor that is part of the Rocket Chip Generator from UC Berkeley [3].

Table 1: Neural Networks for NVDLA Evaluation

Model	Dataset	Layers	Input	Model Size	frames/sec @50MHz
LeNet	MNIST	9	1x28x28	1.7 MB	3.8
Convnet	CIFAR-10	13	3x32x32	572 KB	4.5
SimpleNet	MNIST	44	1x28x28	21 MB	1.3
ResNet-50	ILSVRC2012	229	3x224x224	98 MB	0.4

5 EXPERIMENTAL EVALUATION

Although the NVDLA is a highly configurable accelerator, the NVDLA Compiler supports only two configurations: *NVDLA full* and *NVDLA small*. To test and evaluate the integration of NVDLA in ESP, we use the NVDLA small, which has an 8-bit integer precision, 64 multiply-and-accumulate units, and a 64-bit AXI channel.

Table 1 reports the four neural networks for image classification that we use in these experiments together with their characteristics. For each network, starting from the *Caffe* model and topology specified in *prototxt* format, we generate a calibration table needed for adjusting the network model (trained in full precision) to work at the 8-bit precision of NVDLA small. We then leverage the NVDLA software stack and feed model, topology and calibration table to the NVDLA compiler, which produces an *NVDLA Loadable* containing the layer-by-layer information to configure NVDLA. At runtime, the NVDLA User Mode Driver loads Loadable and input images, and it submits inference jobs to the NVDLA Kernel Mode Driver.

SoC Configuration. Once integrated in ESP, an accelerator can be selected with the GUI and instantiated in multiple tiles during the SoC configuration step. We demonstrate the flexibility and the integration capabilities of ESP by generating various SoC architectures that include one processor tile with the Ariane core, and different numbers of memory tiles and third-party accelerator tiles containing NVDLA. The User Mode Driver and the Kernel Mode Driver run on Ariane, which offloads the inference jobs to the NVDLA instances as needed. When selecting multiple memory tiles, ESP automatically partitions the memory hierarchy to leverage the increased off-chip communication bandwidth. Each memory tile contains a DDR-4 interface to a partition of main memory.

FPGA Prototyping. We deployed each SoC on a proFPGA quad Virtex Ultrascale Prototyping System, which mounts Xilinx XCVU440 FPGAs. On this FPGA, the target ESP runs at 50MHz. First, we ran inference jobs on a single NVDLA instance for the networks in Table 1, which reports the average number of frames per second (fps) processed by an SoC configuration with one NVDLA and one memory tile. The performance depends on the network size, varying between 0.4 fps for ResNet50 and 4.5 fps for Convnet. As a reference, a performance of 7.3 fps is reported [36] for the ResNet50 with an ASIC implementation of NVDLA Small running at a clock frequency of 1GHz, which is twenty time faster than ours.

Performance can be improved by parallelizing the execution of large batches of images across multiple instances of the NVDLA. With ESP, it is easy to explore the design space of possible SoC configurations by tuning the number of NVDLA instances and memory channels utilized in parallel. Since the User Mode and Kernel Mode Drivers provided for NVDLA currently work with a single NVDLA device, we patched them to enable the simultaneous invocation of multiple NVDLA instances from the Ariane core.

Fig. 4 show the results for four SoC configurations, each with an increasing number of NVDLA instances and memory channels utilized in parallel, processing the MNIST dataset with LeNet network.

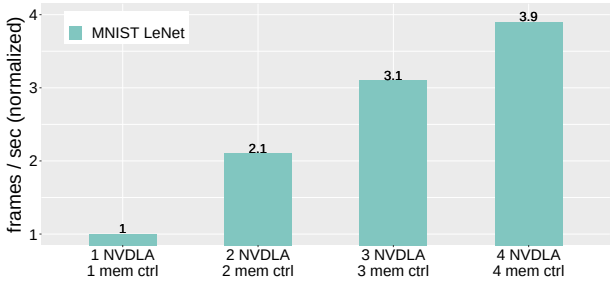

Fig. 4: Scaling NVDLA instances and DDR channels.

The task parallelization delivers an approximately linear increase in performance. For instance, four NVDLA instances with four memory channels bring a 4× speedup for LeNet.

6 RELATED WORK

As the OSH movement has greatly benefited from the success of the RISC-V project [4, 23], the majority of agile flows available to the community revolve around processor-centric systems. For instance, *OpenPiton*, the first SMP Linux booting RISC-V system, supports many Ariane cores with a coherence protocol that extends across multiple chips [7] and can support multi-ISA research [8]. The UC Berkeley team that created RISC-V has released several projects based on their innovative functional RTL language *Chisel* [5]. These open-source systems originate from the *Rocket Chip Generator*, which connects multiple RISC-V cores through a coherent TileLink bus [31]. *Celerity* is a many-core RISC-V SoC that combines open-source RISC-V processors and a single HLS-based neural-network accelerator [16], by leveraging the *Rocket Chip* and its custom co-processor interface *RoCC*. *Rocket Chip* is also at the heart of *FireSim* [28], an FPGA-accelerated RTL simulator that was used to simulate the integration of NVDLA in *Rocket Chip* [18].

All of these open-source frameworks focus mostly on processor cores; reported case studies on the integration of accelerators show that they are either tight to the cores as co-processors [16, 31], or connected with external bus adapters outside of the SoC backbone interconnect [18]. The ESP architecture, instead, implements a distributed system which is inherently scalable, modular and heterogeneous. Processors and loosely-coupled accelerators [15] are given the same importance in the SoC. This system-centric view, as opposed to a processor-centric view, distinguishes ESP from other OSH platforms. Furthermore, the ESP methodology supports different design flows, without imposing any particular tie on the choice of the accelerator specification language and synthesis tool.

7 CONCLUSIONS

We augmented Open ESP with support for the integration of third-party IP blocks, by developing new HW/SW socket interfaces to the ESP architecture and a new design flow to the ESP methodology. While these contributions have a general nature, we demonstrated them by realizing FPGA prototypes of SoCs that feature two major OSH resources: the Ariane processor and the NVDLA accelerator.

Acknowledgments. This research was supported in part by DARPA (C#: HR001118C0122). The views, opinions and/or other findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] Ariane. www.github.com/pulp-platform/ariane.

[2] ARM. AMBA AXI and ACE Protocol Specification. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.html.

[3] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip generator. Technical Report UCB/EECS-2016-17, UC Berkeley, April 2016.

[4] Krste Asanovic and David Patterson. The case for open instruction sets. *Microprocessor Report*, August 2014.

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 1216–1225, 2012.

[6] J. Bai et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2018.

[7] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. OpenPiton+Ariane: the first SMP Linux-booting RISC-V system scaling from one to many cores. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2019.

[8] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, Kunal Gulati, Luca Benini, and David Wentzlaff. BYOC: a "bring your own core" framework for heterogeneous-ISA research. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 699–714, March 2020.

[9] UC Berkeley. TileLink 0.3.3 Specifications. https://docs.google.com/document/d/1Iczcjigc-LUi8QmDPwnAu1kH4Rrt6Kqi1_EUaCrfrk8/pub.

[10] Luca P. Carloni. From latency-insensitive design to communication-based system-level design. *Proceedings of the IEEE*, 103(11):2133–2151, November 2015.

[11] Luca P. Carloni. The case for Embedded Scalable Platforms. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 17:1–17:6, June 2016.

[12] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 20(9):1059–1076, September 2001.

[13] Cobham Gaisler. Leon3 processor. www.gaisler.com/index.php/products/processors/leon3.

[14] Columbia SLD Group. ESP Release. www.esp.cs.columbia.edu, 2019.

[15] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2015.

[16] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald Dreslinski, Christopher Batten, and Michael B. Taylor. The Celerity open-source 511-Core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro*, 38(2):30–41, February 2018.

[17] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Nga Tran, and Zhenbin Wu. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 13(07):P07027–P07027, July 2018.

[18] Farzad Farshchi, Qijing Huang, and Heechul Yun. Integrating NVIDIA deep learning accelerator (NVDLA) with RISC-V SoC on FireSim. *CoRR*, abs/1903.06495, 2019.

[19] Davide Giri, Kuan-Lin Chiu, Giuseppe Di Guglielmo, Paolo Mantovani, and Luca P. Carloni. ESP4ML: platform-based design of systems-on-chip for embedded machine learning. In *Proceedings of the IEEE Conference on Design, Automation, and Test in Europe (DATE)*, March 2020.

[20] Davide Giri, Paolo Mantovani, and Luca P. Carloni. Accelerators & coherence: An SoC perspective. *IEEE Micro*, 38(6):36–45, November 2018.

[21] Davide Giri, Paolo Mantovani, and Luca P. Carloni. NoC-based support of heterogeneous cache-coherence models for accelerators. In *Proceedings of the International Symposium on Networks-on-Chip (NOCS)*, pages 1:1–1:8, October 2018.

[22] Davide Giri, Paolo Mantovani, and Luca P. Carloni. Runtime reconfigurable memory hierarchy in embedded scalable platforms. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 719–726, January 2019.

[23] Samuel Greengard. Will RISC-V revolutionize computing? *Communication of ACM*, 63(5):30–32, April 2020.

[24] Gagan Gupta, Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Kickstarting semiconductor innovation with open source hardware. *IEEE Computer*, 50(6):50–59, June 2017.

[25] Mark Horowitz. Computing's energy problem (and what we can do about it). In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, pages 10–14, February 2014.

[26] IBM. The CoreConnect Bus Architecture. https://www.ibm.comintel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf.

[27] Intel. Avalon Interface Specifications. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf.

[28] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 29–42, 2018.

[29] Keras. https://github.com/fchollet/keras, 2017.

[30] Brucek Khailany, Evgeni Khmer, Rangharajan Venkatesan, Jason Clemons, Joel S. Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Pinckney, Yakun Sophia Shao, Shreesha Srinath, Christopher Torng, Sam (Likun) Xi, Yanqing Zhang, and Brian Zimmer. A modular digital VLSI flow for high-productivity SoC design. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2018.

[31] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, Pi-Feng Chiu, Rimas Avizienis, Brian Richards, Jonathan Bachrach, David Patterson, Elad Alon, Bora Nikolic, and Krste Asanovic. An agile approach to building RISC-V microprocessors. *IEEE Micro*, 36(2):8–20, Mar.-Apr. 2016.

[32] Hung-Yi Liu, Michele Petracca, and Luca P. Carloni. Compositional system-level design exploration with planning of high-level synthesis. In *Proceedings of the IEEE Conference on Design, Automation, and Test in Europe (DATE)*, pages 641–646, March 2012.

[33] Paolo Mantovani, Emilio G. Cota, Christian Pilato, Giuseppe Di Guglielmo, and Luca P. Carloni. Handling large data sets for high-performance embedded applications in heterogeneous systems-on-chip. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES)*, pages 3:1–3:10, October 2016.

[34] Paolo Mantovani, Emilio G. Cota, Kevin Tien, Christian Pilato, Giuseppe Di Guglielmo, Ken Shepard, and Luca P. Carloni. An FPGA-based infrastructure for fine-grained DVFS analysis in high-performance embedded systems. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 157:1–157:6, 2016.

[35] Paolo Mantovani, Giuseppe Di Guglielmo, and L. P. Carloni. High-level synthesis of accelerators in embedded scalable platforms. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 204–211, January 2016.

[36] NVIDIA. NVDLA Primer. www.nvdla.org/primer.html, 2018.

[37] NVIDIA. NVIDIA Deep Learning Accelerator. www.nvdla.org, 2018.

[38] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[39] Luca Piccolboni, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. COSMOS: Coordination of high-level synthesis and memory optimization for hardware accelerators. *ACM Transactions on Embedded Computing Systems*, 16(5s):150:1–150:22, September 2017.

[40] Christian Pilato, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. System-level optimization of accelerator local memory for heterogeneous systems-on-chip. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 36(3):435–448, March 2017.

[41] David Pursley and Tung-Hua Yeh. High-level low-power system design optimization. In *VLSI-DAT*, pages 1–4, April 2017.

[42] HLS4ML. https://fastmachinelearning.org/hls4ml.

[43] Mohandeep Sharma and Dilip Kumar. Wishbone bus architecture - a survey and comparison. *International Journal of VLSI Design and Communication Systems*, 3(2):107–124, April 2012.

[44] Xilinx. The Xilinx Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html.

[45] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-Bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration Systems*, 27(11):2629–2640, November 2019.

[46] Brian Zimmer, Rangharajan Venkatesan, Yakun Sophia Shao, Jason Clemons, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel S. Emer, C. Thomas Gray, Stephen W. Keckler, and Brucek Khailany. A 0.32-128 TOPS, scalable multi-chip-module-based deep neural network inference accelerator with ground-referenced signaling in 16 nm. *IEEE J. of Solid-State Circuits*, 55(4):920–932, April 2020.